UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Refatorações seguras de programas Dafny

Trabalho de Conclusão de Curso

Jonas Bastos Antunes

São Cristóvão – Sergipe

2021

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

Jonas Bastos Antunes

Refatorações seguras de programas Dafny

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Giovanny Fernando Lucero Palma

São Cristóvão – Sergipe

2021

Resumo

Refatoração é uma técnica no qual o código de um programa é modificado, no intuito de melhorar legibilidade e estruturação, porém o comportamento é mantido. Existem ferramentas que realizam refatorações de forma automática, porém muitas não garantem que o comportamento é mantido, ou seja, não realizam refatorações seguras. Este trabalho apresenta a implementação de uma ferramenta que disponibiliza quatro tipos de refatorações seguras para Dafny. As refatorações são seguras no sentido que preservam o comportamento do programa original. O verificador de Dafny é explorado para garantir a segurança das refatorações. Para facilitar o uso para os programadores, também é apresentado neste trabalho uma extensão que integra as refatorações com o editor de texto VSCode. O código-fonte da ferramenta é aberto e pode ser usado como base para adicionar mais refatorações ao catálogo de Dafny.

Palavras-chave: Dafny. refatoração segura. definição formal.

Abstract

Refactoring is a technique in which a program code is modified but preserving its original behaviour, while its structure is improved. Although there are tools that automates refactoring transformations, a fair amount can not ensure the preservation of the orginal behaviour, that is, they do not guarantee safety in the refactorings performed by them. This work presents a safe refactor tool that provides four types of refactorings for the Dafny language. To ensure behaviour preservation, the Dafny verifier assists on the refactoring process. This work also implements a source editor extension for VSCode that provides easy access to the four implemented refactorings. The tool is open source and can be used as foundation for a tool with more refactorings for Dafny.

Keywords: Dafny. safe refactor. formal definition.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

AST	*Abstract Syntatic Tree* (Árvore Sintática Abstrata)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
IC	Iniciação Científica
LSP	*Language Server Protocol* (Protocolo de Servidor de Linguagem)
VSCode	Visual Studio Code

Sumário

1

Introdução

Sistemas computacionais tem sido cada vez mais utilizados em situações críticas onde não se admitem falhas, como o gerenciamento remoto de naves especiais pela NASA (DIVITO; ROBERTS, 1996), gerenciamento de tráfego aéreo (HALL, 1999), cirurgias médicas (SARIN et al., 2010), abastecimento de usinas nucleares (SILVA; SCHIRRU, 2011), plataformas de computação em nuvem (NEWCOMBE et al., 2014), controle de iluminação (DOORNBOS; VERRIET; VERBERKT, 2015) e criptografia de dados (MEADOWS, 2015). No entanto, programas de computadores são apenas um conjunto de instruções escritas por seres humanos que são propícios a cometerem erros. No intuito de oferecer programas mais confiáveis, surgiram linguagens de programação com ferramentas para verificar que os programas são corretos, ou seja, satisfazem suas especificações (WING, 1990). Algumas dessas linguagens incluem *JML* (LEAVENS; BAKER; RUBY, 1998) (SCHOLZ et al., 2011), uma linguagem de especificação para programas em Java, e *Spec#* (BARNETT et al., 2011), uma extensão da linguagem C# que suporta o uso de especificações.

A verificação consiste em uma prova matemática que confronta a implementação com a especificação e é feita usando ferramentas com um grau de automação relativamente alto, porém auxiliado frequentemente pelo desenvolvedor. Em algumas linguagens, tais como Dafny (MICROSOFT, 2009), as especificações são descritas formalmente com pré e pós-condições. O desenvolvedor pode participar das provas adicionando invariantes de laços e assertivas.

Desenvolver sistemas utilizando uma linguagem formal pode se tornar uma tarefa árdua. Adicionar assertivas e invariantes no programa é um processo que normalmente requer um grande esforço intelectual, por vezes complexo. A linguagem Dafny, desenvolvida pela Microsoft, oferece suporte a especificação formal e ao mesmo tempo auxilia automatizando uma parte significativa de provas que confrontam a especificação com o código. No entanto, a participação do desenvolvedor frequentemente é requerida (LEINO, 2017).

Por outro lado, sistemas computacionais normalmente precisam de manutenção, seja

para corrigir falhas ou melhorar sua qualidade. Porém, o processo de alterar um programa pode adicionar novos erros ao sistema computacional. Para minimizar a introdução de erros, uma técnica utilizada é refatoração. Segundo Fowler (1999), essa técnica consiste em modificar um código-fonte com o intuito de melhorar sua estrutura e legibilidade, mas sem alterar seu comportamento.

Os trabalhos de Opdyke (1990) e Griswold (1992) são pioneiros na área de refatoração. Particularmente, Opdyke observou que certas transformações preservam o comportamento dos programas originais quando algumas propriedades são mantidas. O trabalho dele foi o primeiro a definir as transformações junto com condições que garantem propriedades necessárias para preservação do comportamento. Já Griswold se baseou nas Leis da Programação de Hoare et al. (1987), trabalho que trata programas algebricamente de tal forma que transformações que preservam o comportamento são meras manipulações algébricas de programas. Griswold implementou uma ferramenta que realiza transformações algébricas para para obter refatorações.

No entanto, a efetiva prática de refatoração somente se popularizou com a publicação do livro de Fowler (1999), no qual apresenta-se um catálogo de refatorações para programas orientados a objetos com foco na linguagem Java. Outro fato relevante foi o surgimento da Programação Extrema introduzida por Beck (1999). Um dos pilares dessa metodologia é a refatoração, tendo um importante papel no processo de evolução de código. Porém, ambos trabalhos são pragmáticos ao descrever as refatorações de maneira informal e sugerindo o uso metódico de testes para dar confiabilidade quanto à preservação do comportamento.

Junto com os trabalhos de Fowler (1999) e Beck (1999), também se popularizaram ferramentas que realizam refatoração de forma automática, normalmente presentes em ambientes de desenvolvimento como o *Eclipse* (ECLIPSE FOUNDATION, 2001), *Intellij IDEA* (JETBRAINS S.R.O., 2001) e *VSCode* (MICROSOFT, 2015b). Porém, essas ferramentas não oferecem garantias de que o processo ocorre de forma segura, ou seja, sem mudar o comportamento (GLIGORIC et al., 2013) (SOARES; GHEYI; MASSONI, 2012). Inclusive, podem ser inseridos erros que não existiam anteriormente no programa. Segundo levantamentos de Pinto e Kamei (2013) e Tempero, Gorschek e Angelis (2017), o risco de introdução de erros é um dos motivos pelos quais algumas refatorações automáticas são subutilizadas.

Com relação a linguagens formais, tais como Dafny e Spec#, é notável a ausência de ferramentas de refatoração. Nestas linguagens, refatorações inseguras são inaceitáveis e, talvez por isto, ferramentas clássicas de refatoração não tem sido rapidamente adaptadas para seus ambientes de desenvolvimento.

Este trabalho apresenta uma ferramenta de refatoração segura para a linguagem Dafny. As refatorações são realizadas usando especificações formais, que permitem verificar condições estáticas e dinâmicas das refatorações. Essas especificações são verificadas por um provador de teoremas, que garante que o processo mantém o comportamento original.

Especificamente, neste TCC é proposto uma ferramenta que automatiza quatro refatorações para programas Dafny. As refatorações são seguras no sentido que as transformações correspondentes preservam efetivamente os comportamentos originais. Todas estas refatorações baseiam-se em formulações algébricas apresentadas em Lucero (2015). No entanto, aqui são apresentadas de maneira semelhante como é feito em Opdyke (1990) no sentido de especificar a transformação junto com as condições estáticas e dinâmicas que garantem segurança. Mas, diferentemente de Opdyke (1990), as condições dinâmicas são expressas na própria transformação através da inserção de elementos de especificação dentro dos programas.

Este TCC é uma continuação de trabalhos iniciados dentro de um projeto de Iniciação Científica (IC) no qual foi implementado um protótipo de automação da refatoração *Inline Temp* para Dafny (ANTUNES; LUCERO, 2019). A aplicação de *Inline Temp* elimina uma variável temporária substituindo suas ocorrências pela expressão que a define. O projeto se limitou a implementar apenas a transformação e a verificação de algumas condições estáticas simples que, porém, não são suficientes para garantir que a refatoração é segura. O objetivo principal da IC era fazer um estudo prévio de Dafny assim como também de seu compilador.

Um dos objetivos deste TCC é complementar a implementação de *Inline Temp* verificando todas as condições, tanto estáticas como dinâmicas, que garantem segurança na transformação. Adicionalmente, pretende-se implementar refatorações seguras para *Extract Variable*, duas variantes da *Move Method* e uma extensão para o Visual Studio Code que as integra. A refatoração *Extract Variable* realiza a transformação inversa da *Inline Temp*, já as duas variantes de *Move Method* movem um método de uma classe para outra.

O resto deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta uma breve visão da linguagem Dafny, destacando pontos importantes que influenciaram na implementação das refatorações. O Capítulo 3 descreve as refatorações que foram implementadas neste trabalho, junto com suas definições formais. O Capítulo 4 apresenta a ferramenta de refatoração e seus principais módulos. Por fim, o Capítulo 5 conclui o trabalho e apresenta algumas possibilidades de trabalhos futuros.

2

Linguagem Dafny

Dafny (MICROSOFT, 2009) é um sistema de verificação formal moderno que segue a mesma abordagem de Eiffel (EIFFEL SOFTWARE, 1986) oferecendo uma única linguagem tanto para implementação como para especificação. A linguagem de programação inclui construções necessárias para especificação e composição de provas. A intenção é que os desenvolvedores se sintam confortáveis com uma linguagem e ambiente de programação que seguem os moldes clássicos de linguagens imperativas, mas ao mesmo tempo são encorajados a pensar sobre a correção dos programas.

O que diferencia Dafny de linguagens de programação comuns é que ela é projetada para raciocínio formal. Para isto, ela oferece construções de especificação tais como assertivas, invariantes e variantes de laços, pré e pós-condições de métodos, dentre outros. Adicionalmente, Dafny suporta a construção de provas de teoremas e lemas seguindo um paradigma simples e usando para isto as mesmas construções da linguagem de programação.

A linguagem Dafny suporta programação imperativa e funcional. Adicionalmente conta com suporte à programação baseada em objetos, pois oferece classes mas não têm herança nem polimorfismo de herança. O Código 1 ilustra algumas das construções de programação imperativa disponíveis na linguagem através da implementação de uma busca sequencial em um *array* de inteiros.

Diferente das linguagens convencionais, em Dafny não é permitido atribuir a referência nula a variáveis do tipo objeto, exceto se a declaração da variável utilizar o modificador "?". Além disso, expressões não podem conter chamadas de métodos. Outra diferença notável é a verificação dinâmica dos tipos, onde o compilador garante que erros de execução não ocorrerão, incluindo acesso fora dos limites de um *array* ou laços que não finalizam. No Código 1, o compilador infere que o laço `while` finaliza quando a expressão decrescente `arr.Length - i` chega em `0`. Da mesma forma, como `i` nunca assume um valor acima do tamanho do *array*, todos os acessos ao *array* são feitos com índices válidos. A verificação dessas condições é realizada pelo *SMT*

Código 1 – Exemplo de busca sequencial em Dafny.

```
1  method Search(arr: array<int>, value: int)
2  returns (index: int)
3  {
4      var i := 0;
5      while (i < arr.Length && arr[i] != value)
6      {
7          i := i + 1;
8      }
9
10     index := if 0 <= i < arr.Length then i else -1;
11 }
```

Solver Z3 (MICROSOFT, 2008).

2.1 Especificação formal

A linguagem se torna útil para raciocínio formal ao adicionar trechos de especificação nos programas, como ilustrado no Código 2, obtido acrescentando especificações à implementação dada no Código 1. Os métodos foram especificados com pré e pós-condições, representados por `requires` e `ensures` respectivamente. No exemplo, a pré-condição estabelece que o *array* deve possuir pelo menos um elemento. A primeira pós-condição, verificada pela função `foundIndexIsCorrect()`, estabelece que um índice válido retornado representa um elemento do *array* com o mesmo valor de `value`. Já a segunda pós-condição estabelece que não existe nenhum elemento no *array* com o valor `value` caso o índice retornado seja inválido.

Certas pós-condições se tornam inconclusivas pelo verificador de Dafny quando o programa contém laços, onde o fluxo do programa é decidido em tempo de execução. Para auxiliar na validação dessas especificações, a linguagem oferece invariantes, condições que são verificadas cada vez que a expressão de condição do laço é verificada. No Código 2 foram especificadas duas invariantes, nas linhas 9 e 10. A primeira garante que o índice `i` sempre terá valor igual ou inferior ao tamanho do *array*. Já a segunda é uma delimitação da pós-condição definida na linha 5, no qual especifica que todos os índices inferiores a `i` não possuem o valor `value`.

Além das cláusulas para pré, pós-condições e invariantes de laço, Dafny disponbiliza assertivas e o modificador *ghost* para especificações. Uma assertiva é um predicado que deve ser cumprido em certo ponto de um método. Já o modificador *ghost* informa ao compilador que uma determinada declaração será utilizada apenas nas especificações formais, ou seja, não será utilizada na execução do programa. Certas declarações são marcadas automaticamente com `ghost`, como funções. Quando o compilador Dafny gera o programa final, toda a parte relativa a especificações é descartada. Por exemplo, no Código 2 nenhum código é gerado para as cláusulas

Código 2 – Exemplo de uma busca sequencial em Dafny com especificação formal.

```
method Search(arr: array<int>, value: int)
returns (index: int)
requires arr.Length > 0;
ensures foundIndexIsCorrect(arr, index, value)
ensures (0 > index || index >= arr.Length) ==> forall i :: 0 <= i <
    arr.Length ==> arr[i] != value
{
    var i := 0;
    while (i < arr.Length && arr[i] != value)
    invariant i <= arr.Length
    invariant forall x :: 0 <= x < i ==> arr[x] != value
    {
        i := i + 1;
    }

    index := if 0 <= i < arr.Length then i else -1;
}

function foundIndexIsCorrect(arr: array<int>, index: int, value:
    int): bool
reads arr
{
    0 <= index < arr.Length ==> arr[index] == value
}
```

`requires`, `ensures`, `invariant` e para a função `ghost foundIndexIsCorrect()`, sendo esse último usado apenas nas pós-condições do método `Search()`.

2.2 Orientação a objetos

Dafny também oferece suporte limitado a programação orientada a objetos. O Código 3 define uma classe `Counter` que implementa um contador com métodos para incrementar, decrementar e obter seu valor. A sintaxe de definição dos atributos é similar a definição de variáveis, com a diferença que não pode ser definido um valor inicial. Os métodos e funções também possuem sintaxe igual a sua contraparte imperativa, porém neste caso os métodos podem fazer referência à variável `this`, a qual representa o objeto atual. Por fim, o construtor é definido pelo método especial `constructor()`.

Apesar de útil, o suporte para orientação a objetos tem as suas restrições. Algumas delas incluem:

- Ausência de herança, inviabilizando a criação de subclasses a partir de uma classe pai;

- Todas as definições, métodos, funções e atributos de uma classe são públicas. Ou seja, não existe encapsulamento;
- Os atributos não podem ser estáticas, constantes ou somente-leitura;
- Ausência de sobrecarga de métodos.

Código 3 – Exemplo de uma classe em Dafny representando um contador. Fonte: Herbert, Leino e Quaresma (2011).

```
1  class Counter
2  {
3      ghost var Value: int;
4      var incs: int;
5      var decs: int;
6
7      function Valid(): bool
8          reads this;
9      {
10         Value == incs - decs
11     }
12
13     constructor()
14         ensures Valid();
15         ensures Value == 0;
16     {
17         incs, decs, Value := 0, 0, 0;
18     }
19
20     method GetValue() returns (x: int)
21         requires Valid();
22         ensures x == Value;
23     {
24         x := incs - decs;
25     }
26
27     method Inc()
28         requires Valid();
29         modifies this;
30         ensures Valid();
31         ensures Value == old(Value) + 1;
32     {
33         incs, Value := incs + 1, Value + 1;
34     }
35
36     method Dec()
37         requires Valid();
38         modifies this;
39         ensures Valid();
40         ensures Value == old(Value) - 1;
41     {
42         decs, Value := decs + 1, Value - 1;
43     }
44 }
```

3

Refatorações Seguras em Dafny

Este capítulo apresenta a definição das refatorações implementadas neste trabalho. As definições são formais e são adaptações das refatorações descritas em Lucero (2015). Para cada definição apresenta-se um exemplo ilustrativo da transformação correspondente.

3.1 Inline Temp

A refatoração *Inline Temp* consiste na substituição das ocorrências de uma variável pela expressão que define seu valor. Normalmente é utilizada como preparação para aplicar refatorações mais complexas, como a *Replace Temp with Query* e *Extract Method* (Fowler, 1999). A Figura 1 exemplifica seu uso quando aplicado na variável `base` substituindo todas suas ocorrências por `39.00 + 4.99 + 6.99`.

Figura 1 – Exemplo da aplicação da refatoração *Inline Temp*.

```
method PriceWithTaxes(price: real)
returns (finalPrice: real)
{
  var base := 39.00 + 4.99 + 6.99;
  var t:= 120.0;
  return base * price + t;
}
```

```
method PriceWithTaxes(price: real)
returns (finalPrice: real)
{
  var t:= 120.0;
  return (39.00 + 4.99 + 6.99)
         * price + t;
}
```

Definição 1 - Inline Temp

$$\texttt{var}\ t := e;\ c \Rightarrow c'$$

onde:

- *var* $t := e$ é a atribuição da expressão e na variável temporária t;
- c é uma sequência de comandos após a atribuição de t;
- c' é a sequência c com todas as ocorrências da variável t substituídas pela expressão e.

A Definição 1 formaliza a refatoração *Inline Temp*. Para a refatoração ser válida, as seguintes condições devem ser cumpridas:

- Condição estática: a variável t deve ser atribuída uma única vez;
- Condição dinâmica: a expressão e deve se manter constante ao longo de c.

Para especificar a condição dinâmica em Dafny, basta adicionar especificação que, na primeira atribuição a `t`, salve seu valor em uma variável fantasma (*ghost*) e então verifique se, antes de cada uso, a expressão de `t` mantém seu valor inicial. A Figura 2 exemplifica as obrigações de prova inseridas para satisfazer a condição dinâmica requerida para a refatoração do código descrito na Figura 1.

Figura 2 – Exemplo de especificação necessária para aplicar a refatoração *Inline Temp*.

```
method PriceWithTaxes(price: real)
returns (finalPrice: real)
{
  var base := 39.00 + 4.99 + 6.99;

  var t:= 120.0;
  return base * price + t;
}
```

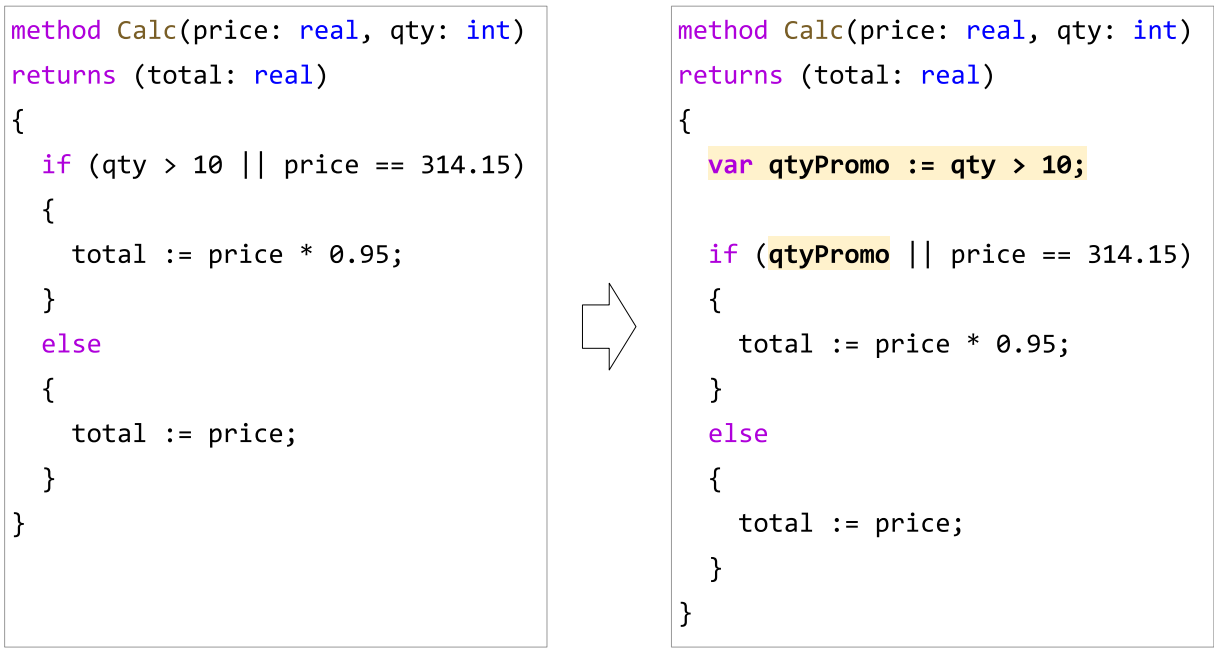
```
method PriceWithTaxes(price: real)
returns (finalPrice: real)
{
  var base := 39.00 + 4.99 + 6.99;
  ghost var base__ghost := base;
  var t:= 120.0;
  assert base__ghost == 39.00 + 4.99
         + 6.99;
  return base * price + t;
}
```

3.2 Extract Variable

A refatoração *Extract Variable* introduz uma variável nova para representar uma dada expressão. Cada ocorrência da expressão é substituída por esta variável. É o processo inverso da refatoração *Inline Temp*, descrita na seção 3.1. Normalmente essa refatoração é utilizada para melhorar a legibilidade do código, especialmente em expressões que aparecem em estruturas condicionais, ou para facilitar a aplicação da refatoração *Extract Method* (Fowler, 1999).

A Figura 3 exemplifica a aplicação da refatoração. A expressão `qty > 10` é extraída e passa a ser representada pela nova variável `qtyPromo`. Todas as ocorrências de `qty > 10` são substituídas por `qtyPromo`.

Figura 3 – Exemplo da aplicação da refatoração *Extract Variable*.



Definição 2 - Extract Variable

$$c \Rightarrow \texttt{var}\ t := e;\ c'$$

onde:

- c é uma sequência de comandos;
- *var* $t := e$ é a atribuição da expressão e na nova variável temporária t;
- c' é a sequência c com todas as ocorrências da expressão e substituídas pela variável t.

A Definição 2 formaliza a refatoração *Extract Variable*. Para a equação ser mantida, as seguintes condições devem ser cumpridas:

- Condição estática: a variável *t* não deve estar previamente declarada no mesmo escopo em que é inserida;
- Condição dinâmica: a expressão *e* deve se manter constante ao longo de *c*.

Para especificar a condição dinâmica em Dafny, basta criar a variável fantasma `t` inicializada com `e` e verificar se as ocorrências da expressão `e` ainda mantém o mesmo valor de `t`. A Figura 4 exemplifica essa situação.

Figura 4 – Exemplo de especificação necessária para aplicar a refatoração *Extract Variable*.

```
method Calc(price: real, qty: int)
returns (total: real)
{
  if (qty > 10 || price == 314.15)
  {
    total := price * 0.95;
  }
  else
  {
    total := price;
  }
}
```

```
method Calc(price: real, qty: int)
returns (total: real)
{
  ghost var promo_ght := qty > 10;

  assert promo_ght == (qty > 10);
  if (qty > 10 || price == 314.15)
  {
    total := price * 0.95;
  }
  else
  {
    total := price;
  }
}
```

3.3 Move Method

A refatoração *Move Method* transfere um método de uma classe para outra. Normalmente essa refatoração é usada para aumentar a coesão das classes, uma vez que é possível transferir o método para uma classe onde é mais usado e faça mais sentido. Também pode diminuir o acoplamento do projeto caso haja muitas interações entre as classes envolvidas.

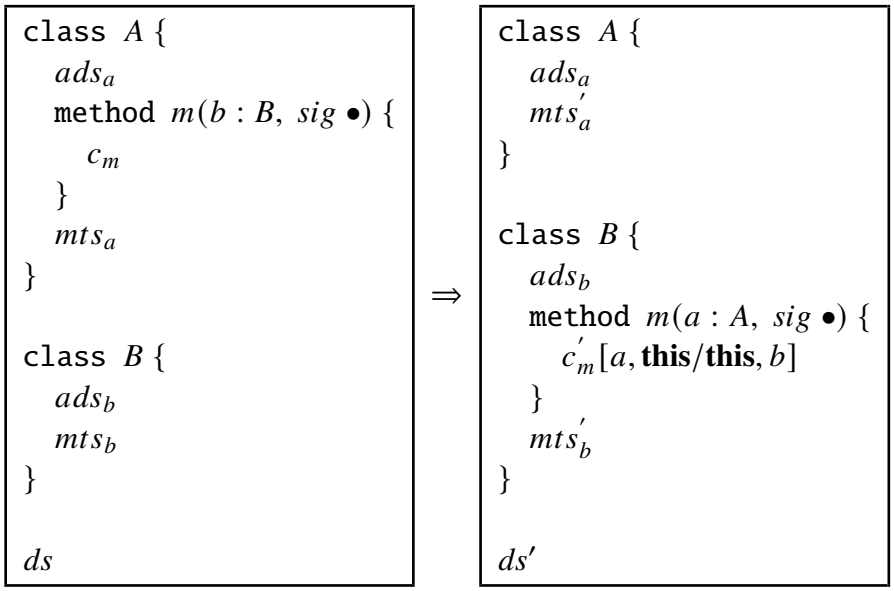
Figura 5 – Exemplo da aplicação da refatoração *Move Method*.

```
class Logger
{
  method Log(message: string)
  {
    print(message);
  }
}

class Shape
{
  var x: string;
  var y: string;

  method Draw(logger: Logger)
  {
    LogDrawing(logger);
  }

  method LogDrawing(logger: Logger)
  {
    var msg := "Shape drawn at "
               + x + ":" + this.y;
    logger.Log(msg);
  }
}
```

```
class Logger
{
  method Log(message: string)
  {
    print(message);
  }

  method LogDrawing(shape: Shape)
  {
    var msg := "Shape drawn at "
               + shape.x + ":"
               + shape.y;
    this.Log(msg);
  }
}

class Shape
{
  var x: string;
  var y: string;

  method Draw(logger: Logger)
  {
    logger.LogDrawing(this);
  }
}
```

A Figura 5 exemplifica esta refatoração. Neste exemplo, o método `LogDrawing()` da classe `Shape` é movido para a classe `Logger`. O alvo original, representado por `this` na classe `Shape`, passa a ser o argumento `logger`. Assim, a transformação realiza as correspondentes renomeações.

Definição 3 - Move Method



onde:

- ads_a são as declarações dos atributos da classe A;
- c_m é o corpo do método m;
- $c'_m[a, \textbf{this}/\textbf{this}, b]$ é o corpo do método m onde a substitui todas as referências a **this** e **this** substitui todas as referências a b. Referências implícitas a **this** também são substituídas;
- mts_a são as declarações dos outros métodos da classe A além de m;
- mts'_a são as declarações mts_a com as referências atualizadas pós-refatoração;
- ds são declarações de variáveis, métodos e outras classes do programa além de A e B;
- ds' são as declarações ds com as referências atualizadas pós-refatoração.

A única condição necessária para garantir que a transformação é válida é que m não esteja declarado como método em B. Observe que esta é uma condição apenas estática.

3.4 Move Method to Associated Class

A refatoração *Move Method to Associated Class* transfere um método para a classe de um atributo da classe origem. Similar ao *Move Method*, essa refatoração é utilizada para aumentar a coesão e diminuir o acoplamento das classes do projeto.

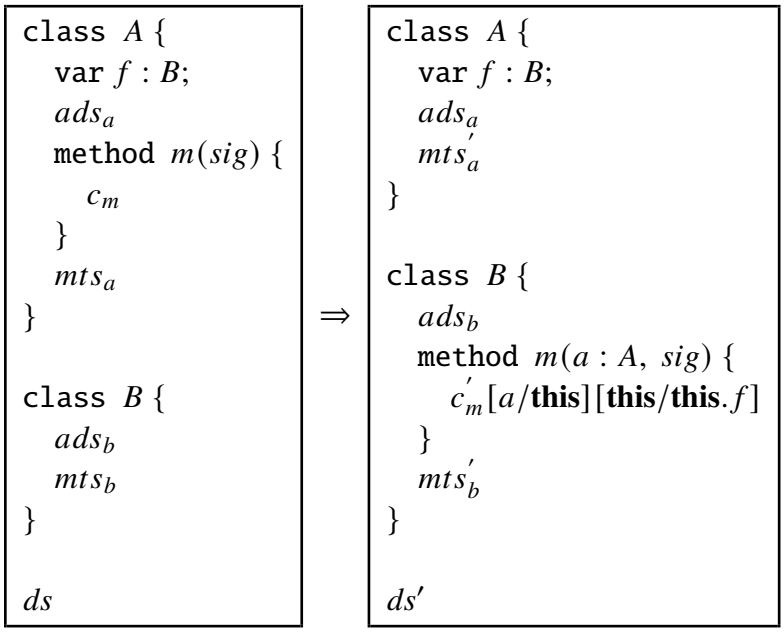
A Figura 6 exemplifica esta refatoração transferindo o método `LogDrawing()` da classe `Shape` para a classe do atributo `logger`, chamada `Logger`. Renomeações necessárias são realizadas pela transformação assim como adequações das chamadas de `LogDrawing()` para ficar compatível com a nova assinatura.

Figura 6 – Exemplo da aplicação da refatoração *Move Method to Associated Class*.

```
class Logger {
  method Log(message: string) {
    print(message);
  }
}

class Shape
{
  var x: string;
  var y: string;
  var logger: Logger;

  constructor(logger: Logger) {
      this.logger := logger;
  }

  method Draw() {
    LogDrawing();
  }

  method LogDrawing() {
    var msg := "Shape drawn at "
               + x + ":" + this.y;
    logger.Log(msg);
  }
}
```

```
class Logger {
  method Log(message: string) {
    print(message);
  }

  method LogDrawing(sp: Shape) {
    var msg := "Shape drawn at "
               + sp.x + ":" + sp.y;
    this.Log(msg);
  }
}

class Shape {
  var x: string;
  var y: string;
  var logger: Logger;

  constructor(logger: Logger) {
      this.logger := logger;
  }

  method Draw() {
    logger.LogDrawing(this);
  }
}
```

Definição 4 - Move Method to Associated Class



onde:

- f é um atributo da classe A para qual o método m será movido;
- ads_a são as declarações dos atributos da classe A;
- c_m é o corpo do método m;
- $c'_m[a/\textbf{this}][\textbf{this}/\textbf{this}.f]$ é o corpo do método m onde a substitui todas as referências a **this** e **this** substitui todas as referências a **this**.f. Referências implícitas a **this**, incluindo **this**.f, também são substituídas;
- mts_a são as declarações dos outros métodos da classe A além de m;
- mts'_a são as declarações mts_a com as referências atualizadas pós-refatoração;
- ds são declarações de variáveis, métodos e outroas classes do programa além de A e B;
- ds' são as declarações ds com as referências atualizadas pós-refatoração.

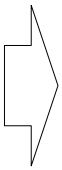
Para a refatoração ser válida e preservar o comportamento, as seguintes condições devem ser cumpridas:

- Condição estática: m não está declarado na classe de destino;
- Condição dinâmica: f se mantém constante ao longo de c_m.

Para expressar a condição dinâmica, adiciona-se no método m original uma especificação que salve no início do método o valor do atributo f em uma variável fantasma e que, imediatamente antes de cada uso de f, assegure mediante uma assertiva que o valor de f se mantém inalterado, sempre igual ao da variável fantasma. A Figura 7 exemplifica a especificação inserida no código descrito na Figura 5 que deverá ser verificado para validar a refatoração.

Figura 7 – Exemplo de assertivas necessárias para aplicar a refatoração *Move Method to Associated Class*.

```
class Logger {
  method Log(message: string) {
    print(message);
  }
}

class Shape {
  var x: string;
  var y: string;
  var logger: Logger;

  constructor(logger: Logger) {
      this.logger := logger;
  }

  method Draw() {
    LogDrawing();
  }

  method LogDrawing() {
    var msg := "Shape drawn at "
               + x + ":" + this.y;
    logger.Log(msg);
  }
}
```



```
class Logger {
  method Log(message: string) {
    print(message);
  }
}

class Shape {
  var x: string;
  var y: string;
  var logger: Logger;

  constructor(logger: Logger) {
      this.logger := logger;
  }

  method Draw() {
    LogDrawing();
  }

  method LogDrawing() {
    ghost var log_ghost := logger;
    var msg := "Shape drawn at "
               + x + ":" + this.y;
    assert logger == log_ghost;
    logger.Log(msg);
  }
}
```

4

Implementação das Refatorações

Neste capítulo será apresentada a implementação das refatorações descritas no Capítulo 3. A seção 4.1 apresenta uma visão geral da arquitetura do compilador do Dafny e dos módulos utilizados na implementação das refatorações. A seção 4.2 demonstra a arquitetura da implementação e suas classes essenciais. Por fim, a Seção 4.3 apresenta a integração da implementação com o *VSCode*.

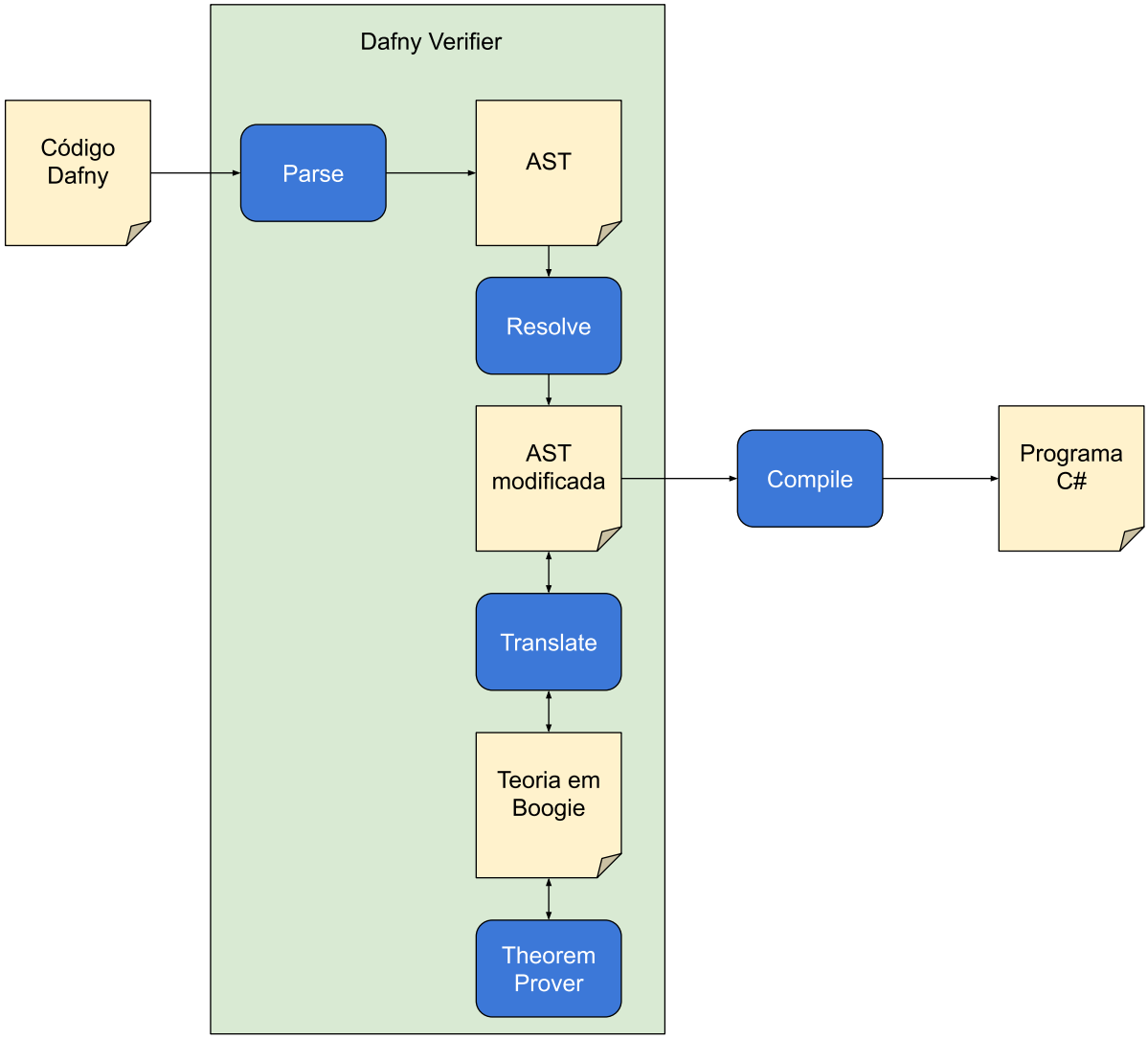
4.1 Compilador Dafny

Uma visão geral da arquitetura do compilador de Dafny é ilustrada na Figura 8. Inicialmente, o módulo *Parse* transforma o código-fonte em Dafny em uma árvore sintática abstrata (*Abstract Syntactic Tree*, AST), uma estrutura de dados comumente utilizada por compiladores para representar um programa. Em seguida, o módulo *Resolve* realiza a análise semântica estática, incluindo verificação de tipos, e então adiciona atributos semânticos aos nós da AST.

Antes de transformar o programa Dafny em um programa C# ou executável binário, é necessário verificar se a AST representa um programa Dafny válido que respeita todos elementos da especificação tais como assertivas, invariantes, pré-condições de métodos e funções. Para isso, o módulo *Translate* traduz a AST para código Boogie (MICROSOFT, 2005), uma linguagem de representação intermediária especializada para verificação formal. Em seguida, o código Boogie é verificado pelo módulo *Theorem Prover* através da geração de obrigações de prova as quais são submetidas ao *SMT-Solver* Z3 (MICROSOFT, 2008). Caso o módulo *Theorem Prover* não consiga realizar alguma das provas, o processo de compilação é abortado. Por fim, após verificar que o código é válido, a AST é traduzida pelo módulo *Compile* para um programa C# ou executável binário.

Dois componentes do compilador são utilizados na implementação deste TCC. O primeiro deles é *Parse*, o gerador da árvore sintática abstrata. A partir da AST, é possível fazer as análises

Figura 8 – Arquitetura do compilador Dafny.



necessárias para realizar a refatoração.

Normalmente, uma refatoração requer tanto análises estáticas quanto verificações de condições. Nesta implementação, as primeiras são realizadas em cima da AST, como seria de esperar. Já para verificação apresentam-se algumas opções. Do ponto de vista de eficiência, modularização e integração com o compilador Dafny, talvez o mais adequado seria gerar, a partir da AST, código Boogie contendo as obrigações de prova das condições a serem verificadas pelo Z3. No entanto, para este TCC, esta opção não é atraente pois demanda o estudo de Boogie e Z3. Então, optou-se por adicionar as condições como assertivas dentro do programa Dafny a ser refatorado. Para isto, apresentam-se como alternativas adicionar as assertivas na AST ou no próprio programa fonte. Optou-se pela segunda alternativa, pois a AST gerada pelo compilador Dafny não permite alterações *in loco*.

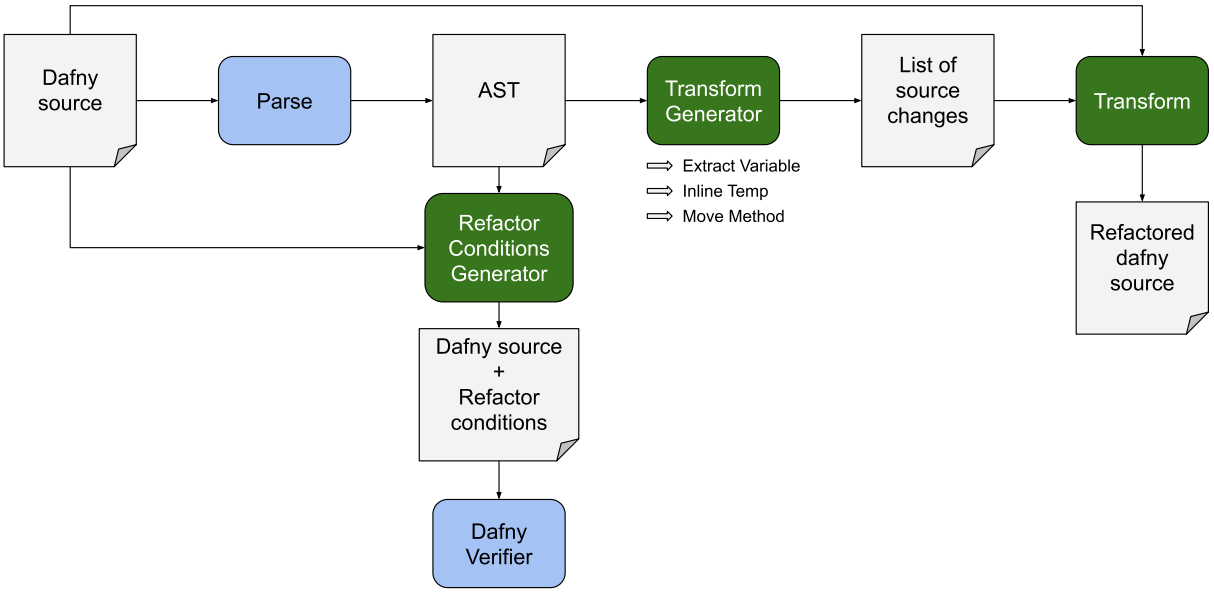
O segundo componente do compilador utilizado pela ferramenta de refatoração é o

provador de teoremas *Theorem Prover*, usado para verificar as condições necessárias para que uma determinada refatoração seja segura. O verificador requer como argumento uma representação intermediária em Boogie, que pode ser gerada a partir da AST modificada na etapa *Resolve*. A ferramenta deste TCC usa o verificador de forma indireta, através do processo de compilação de um programa Dafny.

O uso indireto do provador ocorre da seguinte forma: inicialmente, para adicionar especificações das condições necessárias para a segurança da refatoração no programa a ser refatorado, é criada uma lista onde cada um dos seus elementos contém código de especificação junto com a posição em que será inserido dentro do programa fonte. Em seguida, a lista de inserções é aplicada temporariamente no código-fonte. Por fim, o compilador Dafny é executado até a etapa de verificação do código, devidamente configurado através da opção */compile:0*. Se o processo de compilação for executado com sucesso, então o programa pode ser refatorado com segurança. Caso contrário, não será possível garantir a segurança da refatoração.

4.2 Ferramenta de refatoração

Figura 9 – Arquitetura da ferramenta de refatoração.



A Figura 9 ilustra uma visão geral da arquitetura da ferramenta, que foi implementada na linguagem C#. Inicialmente, o código-fonte é transformado para uma árvore sintática abstrata pela etapa *Parse*. Antes de aplicar a transformação requerida pela refatoração, é necessário verificar a validade das condições estáticas e dinâmicas do programa. Para isso, a etapa *Refactor Conditions Generator* analisa a AST e gera o programa que contém as condições exigidas para a segurança da refatoração, como exemplificado na Figura 2 para o caso da refatoração *Inline Temp*. Esse programa com assertivas é verificado pelo verificador do Dafny no passo *Dafny Verifier*. Se

o verificador conseguir provar que as condições são cumpridas, então a refatoração prossegue. Do contrário, o processo é abortado.

Em seguida, a etapa *Transform Generator* gera a lista de mudanças que serão aplicadas no programa fonte para derivar a refatoração, como exemplificado na Figura 10 para o mesmo cenário da Figura 2. Por fim, as mudanças são aplicas no código-fonte no passo *Transform* e o código-fonte refatorado é gerado.

Figura 10 – Exemplo de mudanças necessárias no código para aplicar a refatoração *Inline Temp*. O trecho riscado significa uma remoção e o trecho sem riscos com fundo destacado significa uma adição.

List<SourceEdit>		
start	end	content
69	101	""
132	136	"(39.0 + 4.99 + 6.99)"

```
method PriceWithTaxes(price: real)
returns (finalPrice: real) {
  ~~var base := 39.00 + 4.99 + 6.99;~~
  var t:= 120.0;
  return ~~base~~ (39.0 + 4.99 + 6.99) * price + t;
}
```

4.2.1 RefactorStep<TState>

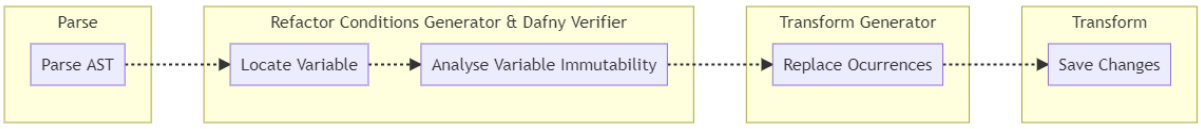
O processo de refatoração normalmente envolve uma série de passos. Por exemplo, a refatoração *Inline Temp* precisa:

- Transformar o código-fonte em AST;
- Localizar a variável;
- Verificar se a variável é constante;
- Substituir todas as ocorrências da variável pela sua expressão;
- Salvar as mudanças no código-fonte.

Algumas etapas podem ser reaproveitados por outras refatorações, como a transformação de código em AST. Para manter o projeto escalável e flexível, foi implementada a classe base

`RefactorStep<TState>`, seguindo o padrão de projeto *Chain of Responsability* (GAMMA et al., 1995). Cada refatoração implementada neste trabalho é representada por uma sequência de etapas, instancias de subclasses de *RefactorStep*. A Figura 11 ilustra a divisão da refatoração *Inline Temp* em etapas, onde cada etapa está agrupada pelos módulos que utiliza da arquitetura descrita na Figura 9.

Figura 11 – Diagrama de fluxo simplificado representando os passos da refatoração *Inline Temp*.



O Código 4 apresenta a implementação da classe base. Cada etapa contém uma referência para a etapa seguinte através da propriedade `next`, similar a uma lista encadeada. Caso a referência seja nula, significa que essa é a última etapa da cadeia de responsabilidade.

A classe contém um único método, chamado `Handle()`. Esse método tem um único parâmetro `state` que implementa `IRefactorState`, uma interface que guarda a AST, argumentos da linha de comando, uma lista de mudanças que devem ser aplicadas no código e uma lista de erros que foram registrados ao longo do processo de refatoração. Cada etapa pode manipular e salvar dados no estado comum `state` para que etapas subsequentes utilizem essas informações. Por exemplo, a etapa *Parse* carrega a árvore sintática em `state.Program`. Com isso, os passos *Locate Variable* e *Replace Ocurrences* tem acesso a AST do programa.

No método `Handle()` devem ser executadas as ações de análise e manipulação do programa. No fim desse método, deve ser chamado o próximo passo a ser executado, caso nenhum erro tenha acontecido. O objeto `state` é passado como argumento para o próximo passo em `next.Handle(state)`.

Código 4 – Implementação da classe *RefactorStep<TState>*.

```
1  public abstract class RefactorStep<TState> where TState :
       IRefactorState
2  {
3      public RefactorStep<TState> next;
4
5      public virtual void Handle(TState state)
6      {
7          if (state.Errors.Count > 0) return;
8          next?.Handle(state);
9      }
10 }
```

O Código 5 ilustra um dos passos implementados na refatoração *Inline Temp* deste

trabalho, responsável por encontrar a variável na AST a partir da posição no qual o *token* da variável está localizada no código-fonte. Primeiramente, é executado o método de busca da variável, definido na classe `VariableLocator`. Depois, é analisado se alguma variável foi encontrada. Caso não tenha sido encontrada, uma mensagem de erro é relatada e a execução da cadeia de responsabilidade é interrompida. Em seguida, a variável é salva no estado compartilhado da refatoração, no qual os próximos passos poderão contar com essa informação para executar suas atividades. Por fim, é chamado a implementação da superclasse, que executa o próximo passo caso nenhum erro tenha sido registrado.

Código 5 – Implementação simplificada da classe `LocateVariableStep<T>`.

```
1  public class LocateVariableStep<TState> : RefactorStep<TState> where
       TState : IInlineTempState
2  {
3      public override void Handle(TState state)
4      {
5          var foundVariable = VariableLocator.Locate(state.Program,
               state.RootSymbolTable, state.UserSelectionPos);
6          if (foundVariable == null)
7          {
8              state.AddError(InlineTempErrorMsg.NotFoundVariable());
9              return;
10         }
11
12         state.InlineVariable = foundVariable;
13
14         base.Handle(state);
15     }
16 }
```

4.2.2 `DafnyVisitor`

Normalmente, cada etapa do processo de refatoração utiliza a AST de alguma forma, seja para análise ou manipulação do programa. Porém a AST é uma estrutura complexa e seu manejo requer a consideração de muitos casos. No intuito de facilitar a manipulação e navegação pela AST, foi implementada a classe base `DafnyVisitor`, seguindo o padrão de projeto *Visitor* (GAMMA et al., 1995). Ela permite percorrer pelos elementos importantes da árvore sintática para realizar as refatorações implementadas nesse trabalho.

O Código 6 apresenta um trecho da assinatura da classe. A classe não possui um método fixo de visitação inicial. Com isso, as subclasses podem começar a navegação a partir de qualquer nó, seja de um objeto `Program`, nó raiz que representa um programa, ou de um `Statement`, por exemplo.

A assinatura pode ser dividida em dois grupos de métodos: `Visit()` e `Traverse()`. Os

Código 6 – Assinatura da classe *DafnyVisitor*.

```
1  public abstract class DafnyVisitor
2  {
3      protected virtual void Visit(Program prog) {}
4      protected virtual void Visit(Statement stmt) {}
5      protected virtual void Visit(VarDeclStmt vds) {}
6      protected virtual void Visit(BlockStmt block) {}
7      // ...
8
9      protected virtual void Visit(Expression exp) {}
10     protected virtual void Visit(BinaryExpr binaryExpr) {}
11     protected virtual void Visit(LiteralExpr literalExpr) {}
12     // ...
13
14     protected virtual void Traverse(List<Statement> stmts) {}
15     protected virtual void Traverse(List<Expression> exprs) {}
16 }
```

métodos `Visit()` recebem um único elemento como argumento e são responsáveis por fazer a visitação nesse elemento. O Código 7 demonstra como funciona a visitação padrão de um elemento `Method` da AST.

Código 7 – Implementação de um típico método `Visit()` da classe `DafnyVisitor`.

```
1  protected virtual void Visit(Method mt)
2  {
3      Visit(mt.Body);
4  }
```

Para elementos genéricos como `Statement`, por exemplo, o método `Visit(Statement)` especializa cada caso através do *casting* de tipos. O Código 8 mostra a implementação deste método. Como um `Statement` tem subclasses específicas, é necessário converter o tipo para uma subclasse antes de realizar alguma ação. Quando é possível converter o tipo, é chamado o método `Visit()` da subclasse correspondente.

Alguns elementos da AST, como por exemplo `ClassDecl`, possuem vários filhos agrupados em uma lista. Visando simplificar a implementação dos métodos `Visit()`, a responsabilidade de manipular essas listas foi transferida para os métodos `Traverse()`. O Código 9 mostra a visitação de uma lista de `Statement`. Basicamente, cada elemento da lista é redirecionado para o visitador de `Statement`, cuja implementação foi apresentada no Código 8.

Além da classe base `DafnyVisitor`, também foi implementado a subclasse base `DafnyVisitorWithNearests`. Ela estende as funcionalidades de `DafnyVisitor` e adicionalmente guarda o token do bloco mais recente visitado, no atributo `nearestTokenBlock`, e o `Statement` mais recente visitado, no atributo `nearestStmt`.

Código 8 – Implementação simplificada de um método `Visit()` da classe `DafnyVisitor` que é focado em realizar conversões de tipos.

```
protected virtual void Visit(Statement stmt)
{
    switch (stmt)
    {
        case VarDeclStmt vds:
            Visit(vds);
            break;
        case BlockStmt block:
            Visit(block);
            break;
        // ...

        default:
            Traverse(stmt.SubExpressionsList);
            Traverse(stmt.SubStatementsList);
            break;
    }
}
```

Código 9 – Implementação de um típico método *Traverse()* da classe *DafnyVisitor*.

```
protected virtual void Traverse(List<Statement> body)
{
    foreach (var stmt in body)
    {
        Visit(stmt);
    }
}
```

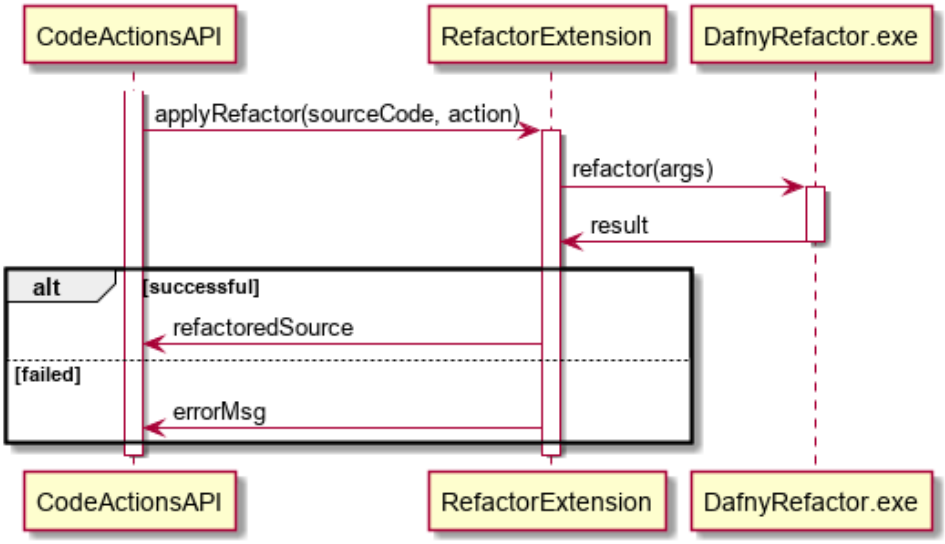
A ordem de visitação implementada por padrão é a *preorder*. Para manter esse comportamento ao estender a classe original, basta chamar a implementação do método da superclasse, `base.Visit(el)` ou `base.Traverse(list)`, após manipular o elemento ou lista. É possível trocar facilmente a ordem de visitação para *postorder*, em um determinado elemento, ao chamar a implementação da superclasse antes de manipular o elemento ou lista. Também é possível implementar outras ordens de visitação, bastando reescrever os métodos que forem necessários.

O Código 10 ilustra o uso de uma classe que estende as funcionalidades do `DafnyVisitor`. Essa classe procura uma variável no programa de acordo com a posição do *token* no código-fonte. Uma vez que a variável é encontrada, o programa devolve o símbolo correspondente da tabela de símbolos. Para simplificar o uso dessa classe, o método estático `VariableLocator.Locate()` é disponibilizado. Ele cria um objeto da classe, realiza a busca e retorna o resultado.

4.3 Extensão do Visual Studio Code (VSCode)

A Figura 12 apresenta um diagrama de sequência do processo de refatoração do ponto de vista do VSCode. Quando o usuário da extensão seleciona um trecho de código e chama a lista de ações, um menu de operações disponíveis aparece no VSCode, como ilustrado na Figura 13. Quando escolhida uma das opções disponibilizadas pela extensão, a respectiva função de refatoração é executada, representada por `applyRefactor()` no diagrama.

Figura 12 – Diagrama de sequência da refatoração do ponto de vista do VSCode.



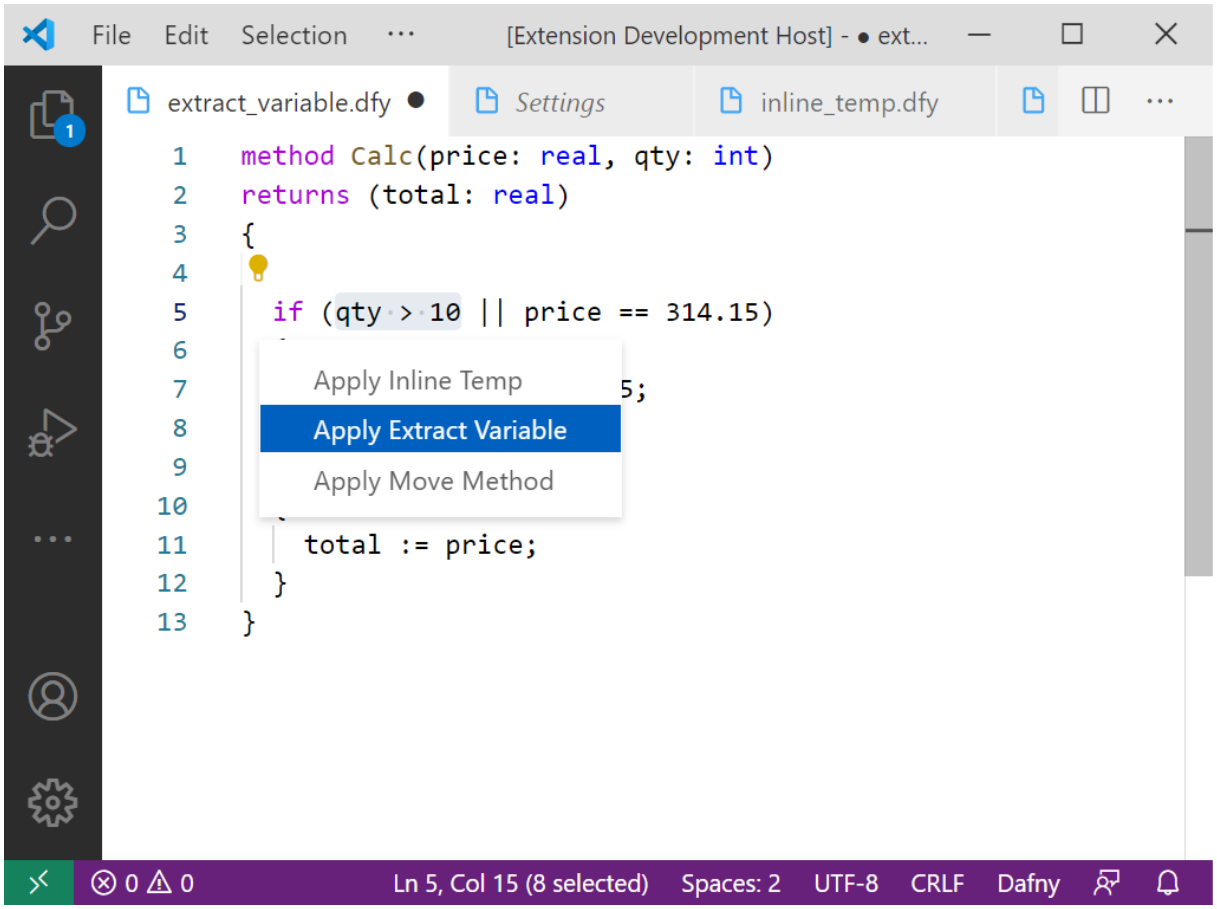
Cada função coleta o código aberto no VSCode, executa a ferramenta de refatoração com o programa coletado e recebe de volta o código refatorado ou uma mensagem de erro. Dependendo da resposta, a extensão atualiza o código-fonte aberto no VSCode com o código refatorado ou exibe uma mensagem de erro.

4.3.1 Limitações

Atualmente, a extensão e a ferramenta de refatoração se comunicam diretamente através dos fluxos padrões de comunicação entre processos, ou seja, o *stdin*, *stdout* e *stderr*. Esse é um tipo de comunicação muito simples, de nível binário, e requer uma integração muito acoplada da ferramenta com a extensão do VSCode.

Idealmente essa integração deveria utilizar um protocolo de alto nível, como o *Language Server Protocol* (LSP) (MICROSOFT, 2015a). Esse protocolo define padrões de mensagens e ações para manipular programas. A IDE é chamada de *cliente* e o processo responsável por analisar e manipular o programa é chamado de *servidor*. Toda a comunicação entre o cliente e servidor acontece em alto nível, sem envolver objetos complexos como a árvore sintática.

Figura 13 – Opções de refatoração da extensão do VSCode.



Ao usar um servidor LSP da mesma linguagem da ferramenta de refatoração, C#, é possível ter acesso a todos os métodos de refatoração já implementados neste trabalho. Em contrapartida, o LSP exige a definição de vários objetos de manipulação das requisições. Esses objetos são complexos e requerem uma integração mais profunda com o compilador do que a utilizada pela ferramenta de refatoração implementada neste trabalho, que basicamente se restringe a utilizar a árvore sintática e o provador de teoremas.

Código 10 – Implementação simplificada da classe *VariableLocator*.

```
1  public class VariableLocator : DafnyVisitorWithNearests
2  {
3      private readonly int _position;
4      private readonly Program _program;
5      private readonly ISymbolTable rootSymbolTable;
6      public IInlineVariable foundDeclaration;
7
8      private ISymbolTable CurSymbolTable =>
           rootSymbolTable.FindSubTableByToken(nearestBlockToken);
9
10     private void Execute()
11     {
12         Visit(_program);
13     }
14
15     protected override void Visit(VarDeclStmt vds)
16     {
17         foreach (var local in vds.Locals)
18         {
19             Analyze(local.token);
20         }
21
22         base.Visit(vds);
23     }
24
25     protected override void Visit(NameSegment nameSeg)
26     {
27         Analyze(nameSeg.token);
28     }
29
30     private void Analyze(IToken token)
31     {
32         int start = token.pos;
33         int end = token.pos + token.val.Length;
34         if (start > _position || _position > end) return;
35
36         foundDeclaration =
               CurSymbolTable.LookupVariable(token.value);
37     }
38
39     public static IInlineVariable Locate(Program program,
           ISymbolTable rootSymbolTable, int position)
40     {
41         var locator = new VariableLocator(program, rootSymbolTable,
               position);
42         locator.Execute();
43         return locator._foundDeclaration;
44     }
45 }
```

5
Conclusão

Neste trabalho foi apresentado a implementação de uma ferramenta de refatoração segura para Dafny, usando como base as formalizações descritas por (LUCERO, 2015). Quatro refatorações foram implementadas, sendo elas: *Inline Temp*, *Extract Variable*, *Move Method* e *Move Method to Associated Class*. Além da ferramenta, foi apresentado uma extensão para o VSCode que permite realizar as refatorações diretamente da IDE. Espera-se que este trabalho facilite a aplicação de refatorações em programas Dafny, assim como sirva de base e inspiração para outros trabalhos sobre refatorações seguras, inclusive para outras linguagens. Em particular, com relação a Dafny, a estrutura do projeto desenvolvido aqui é modular e pode ser estendida com novas refatorações.

Na implementação da ferramenta usam-se como componentes módulos do compilador e do verificador de Dafny. Enquanto o verificador se apresentou adequado, do compilador foi reutilizado apenas o *Parser*, principalmente pela restrição da AST gerada não poder ser alterada *in loco*, o que dificulta a implementação das transformações através da modificação da AST. Além disso, o módulo Dafny responsável por transformar a AST em código-fonte gera código que não se corresponde fielmente com o código fonte original quando este não é alterado pela transformação. De fato, o propósito deste gerador é apenas auxiliar na depuração do compilador. Assim, um método alternativo de modificação de programas precisou ser usado, tornando o processo de implementação das refatorações complexo e mais prolongado do que o planejado inicialmente. Particularmente, a transformação da refatoração *Extract Variable* foi a mais trabalhosa de ser implementada e a técnica usada resultou ser ineficiente, apesar da refatoração funcionar corretamente. Um trabalho futuro poderia reimplementar as alterações de código diretamente na AST, apresentando assim uma solução mais elegante e eficiente do que a atual.

Inicialmente foi planejado confirmar a eficácia da ferramenta através de um conjunto de testes. Para isso, foi montado um repositório com códigos de programas Dafny criados

pela comunidade (ANTUNES, 2019). Apesar do repositório formar um conjunto de tamanho significativo de programas Dafny, todos são de caráter acadêmico ou experimentais, e, como regra, bem estruturados. Isto torna o repositório inadequado para testar refatorações, pois não há necessidade delas. Por este motivo, as funcionalidades implementadas pela ferramenta foram testadas apenas com códigos simples ao invés do repositório de códigos.

A integração com o VSCode foi implementada visando mais em simplicidade do que em portabilidade, como explicada na Seção 4.3.1. Idealmente o trabalho deveria utilizar um servidor LSP, tornando a extensão mais eficiente e compatível com outras IDEs que suportam o protocolo. Uma abordagem seria utilizar com base o servidor LSP oficial do Dafny (MICROSOFT, 2020). Porém a ferramenta de refatoração foi implementada em *.NET Framework*, já que essa era a *framework* utilizada na época pelo compilador no início do desenvolvimento deste trabalho. Recentemente, o compilador do Dafny migrou o seu projeto de *.NET Core*, com a primeira versão de prévia lançada em agosto de 2020. O servidor LSP oficial do Dafny teve a sua primeira lançada em novembro de 2020, já utilizando a migração para o *.NET Core*. Para utilizar o servidor LSP do Dafny em conjunto com a ferramenta de refatoração, seria necessário migrar todo o projeto deste TCC para *.NET Core*. Devido a isso, se tornou inviável o uso do LSP neste TCC. Um trabalho futuro poderia migrar a ferramenta de refatoração para o *.NET Core* e implementar um servidor LSP.

Além das refatorações implementadas neste trabalho, outras refatorações estão formalizadas por trabalhos como Lucero (2015), Cornélio, Cavalcanti e Sampaio (2010) e Garrido e Meseguer (2006). Apesar de serem refatorações para outras linguagem, elas podem ser adaptadas para a linguagem Dafny e adicionar mais tipos de refatorações para a ferramenta proposta aqui. Em particular, certas construções da linguagem não foram abordadas, como *traits* e *modules*. Um trabalho futuro incluiria refatorações que lidem com estas construções.

Referências

ANTUNES, J. B. *Dafny code samples*. 2019. Disponível em: <https://github.com/jonasbantunes/dafny-samples>. Acesso em: 26 jan 2021. Citado na página 37.

ANTUNES, J. B.; LUCERO, G. F. P. *Refatorações seguras de programas corretos*. 2019. Disponível em: <https://drive.google.com/open?id=1CzZ3gXaGl5qKoIYMuI0fqx5RS73IPVn1>. Acesso em: 25 fev 2020. Citado na página 10.

BARNETT, M. et al. Specification and verification: the spec# experience. *Communications of the ACM*, ACM New York, NY, USA, v. 54, n. 6, p. 81–91, 2011. Citado na página 8.

Beck, K. *Extreme Programming Explained: Embrace Change*. [S.l.: s.n.], 1999. Citado na página 9.

CORNÉLIO, M.; CAVALCANTI, A.; SAMPAIO, A. Sound refactorings. *Science of Computer Programming*, Elsevier, v. 75, n. 3, p. 106–133, 2010. Citado na página 37.

DIVITO, B. L.; ROBERTS, L. W. Using formal methods to assist in the requirements analysis of the space shuttle gps change request. 1996. Citado na página 8.

DOORNBOS, R.; VERRIET, J.; VERBERKT, M. Robustness analysis for indoor lighting systems. In: *10th International Conference on Systems, Barcelona, Spain*. [S.l.: s.n.], 2015. p. 122. Citado na página 8.

ECLIPSE FOUNDATION. *Eclipse IDE - The Leading Open Platform for Professional Developers*. 2001. Disponível em: <https://www.eclipse.org/eclipseide/>. Acesso em: 27 dec 2020. Citado na página 9.

EIFFEL SOFTWARE. *Eiffel Community*. 1986. Disponível em: <https://www.eiffel.org/>. Acesso em: 25 jan 2021. Citado na página 11.

Fowler, M. *Refactoring: Improving the Design of Existing Code*. [S.l.: s.n.], 1999. Citado 3 vezes nas páginas 9, 16 e 18.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201633612. Citado 2 vezes nas páginas 29 e 30.

GARRIDO, A.; MESEGUER, J. Formal specification and verification of java refactorings. In: IEEE. *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. [S.l.], 2006. p. 165–174. Citado na página 37.

GLIGORIC, M. et al. Systematic testing of refactoring engines on real software projects. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2013. p. 629–653. Citado na página 9.

GRISWOLD, W. G. Program restructuring as an aid to software maintenance. 1992. Citado na página 9.

HALL, A. Using formal methods to develop an atc information system. In: *Industrial-Strength Formal Methods in Practice*. [S.l.]: Springer, 1999. p. 207–229. Citado na página 8.

HERBERT, L.; LEINO, K. R. M.; QUARESMA, J. Using dafny, an automatic program verifier. In: SPRINGER. *LASER Summer School on Software Engineering*. [S.l.], 2011. p. 156–181. Citado 2 vezes nas páginas 5 e 15.

HOARE, C. A. R. et al. Laws of programming. *Communications of the ACM*, ACM New York, NY, USA, v. 30, n. 8, p. 672–686, 1987. Citado na página 9.

JETBRAINS S.R.O. *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. 2001. Disponível em: <https://www.jetbrains.com/idea/>. Acesso em: 06 dec 2020. Citado na página 9.

LEAVENS, G. T.; BAKER, A. L.; RUBY, C. Jml: a java modeling language. In: CITESEER. *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. [S.l.], 1998. p. 404–420. Citado na página 8.

LEINO, K. R. M. Accessible software verification with dafny. *IEEE Software*, IEEE, v. 34, n. 6, p. 94–97, 2017. Citado na página 8.

LUCERO, G. F. P. Algebraic laws for object oriented programming with references. Universidade Federal de Pernambuco, 2015. Citado 4 vezes nas páginas 10, 16, 36 e 37.

MEADOWS, C. Emerging issues and trends in formal methods in cryptographic protocol analysis: Twelve years later. In: *Logic, rewriting, and concurrency*. [S.l.]: Springer, 2015. p. 475–492. Citado na página 8.

MICROSOFT. *Boogie - GitHub*. 2005. Disponível em: <https://github.com/boogie-org/boogie>. Acesso em: 11 nov 2020. Citado na página 25.

MICROSOFT. *The Z3 Theorem Prover*. 2008. Disponível em: <https://github.com/Z3Prover/z3>. Acesso em: 11 nov 2020. Citado 2 vezes nas páginas 12 e 25.

MICROSOFT. *Dafny - GitHub*. 2009. Disponível em: <https://github.com/dafny-lang/dafny>. Acesso em: 11 nov 2020. Citado 2 vezes nas páginas 8 e 11.

MICROSOFT. *Dafny - GitHub*. 2015. Disponível em: <https://github.com/dafny-lang/dafny>. Acesso em: 11 nov 2020. Citado na página 33.

MICROSOFT. *Visual Studio Code - Code Editing. Redefined*. 2015. Disponível em: <https://code.visualstudio.com/>. Acesso em: 11 nov 2020. Citado na página 9.

MICROSOFT. *DafnyLS - GitHub*. 2020. Disponível em: <https://github.com/dafny-lang/language-server-csharp>. Acesso em: 28 dec 2020. Citado na página 37.

NEWCOMBE, C. et al. Use of formal methods at amazon web services. *See http://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf*, 2014. Citado na página 8.

OPDYKE, W. F. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: *Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*. [S.l.: s.n.], 1990. Citado 2 vezes nas páginas 9 e 10.

PINTO, G. H.; KAMEI, F. What programmers say about refactoring tools? an empirical investigation of stack overflow. In: *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*. [S.l.: s.n.], 2013. p. 33–36. Citado na página 9.

SARIN, V. K. et al. *Non-imaging, computer assisted navigation system for hip replacement surgery*. [S.l.]: Google Patents, 2010. US Patent 7,780,681. Citado na página 8.

SCHOLZ, W. et al. Automatic detection of feature interactions using the java modeling language: an experience report. In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. [S.l.: s.n.], 2011. p. 1–8. Citado na página 8.

SILVA, M. H. da; SCHIRRU, R. Optimization of nuclear reactor core fuel reload using the new quantum pbil. *Annals of Nuclear Energy*, Elsevier, v. 38, n. 2-3, p. 610–614, 2011. Citado na página 8.

SOARES, G.; GHEYI, R.; MASSONI, T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 2, p. 147–162, 2012. Citado na página 9.

TEMPERO, E.; GORSCHEK, T.; ANGELIS, L. Barriers to refactoring. *Communications of the ACM*, ACM New York, NY, USA, v. 60, n. 10, p. 54–61, 2017. Citado na página 9.

WING, J. M. A specifier's introduction to formal methods. *Computer*, IEEE, v. 23, n. 9, p. 8–22, 1990. Citado na página 8.